POWER PROGRAMMING

Exploring the WIN16, WIN32, and WIN32S APIs

BY RAY DUNCAN

Everyone agrees that OS/2 1.*x* was a flop, but there's little agreement about *why* it flopped. OS/2's bumpy history has been affected by a vast constellation of personalities, strategic decisions, and market forces both seen and invisible. IBM supporters feel that Microsoft contributed to OS/2's problems first by failing to commit itself wholeheartedly to OS/2 application and device driver development, and then by unilaterally abandoning the operating system when it became financially expedient to do so. Microsoft partisans, on the other hand, claim that IBM doomed OS/2 by insisting on a 286 version, by linking OS/2's destiny with the PS/2, and by inventing a new graphical API instead of simply porting Windows to protected mode and running it on top of the OS/2 kernel. There are those who see the last six years' events as evidence of sinister plots at IBM or Microsoft, and others who see the same events as healthy manifestations of a capitalist economy and a free market.

Clearly, there are no simple explanations for what went wrong, and nearly all of the proposed explanations for OS/2's problems are subject to various interpretations. For example, it's certainly fair to condemn Microsoft for reneging on its commitments to OS/2 developers two years ago when it decided to focus on Windows development. It's equally fair to criticize Microsoft for its decision three years earlier to embrace the new Presentation Manager (PM) API just to gain IBM's endorsement, leaving Windows developers (both inside and outside Microsoft) to twist slowly in the wind. If there's a lesson to be learned here, it's that developers of any stripe need to be wary, because Microsoft never learned how to set long-range goals and then stick to them through good times and bad.

In any event, at least one of the OS/2 theories is about to be publicly tested—the premise that the OS/2 1.*x* PM API was "too big a delta" from the Windows 2.*x* API, that developers couldn't handle the transition, and that consequently OS/2 suffered from a lack of pull-through demand generated by hypothetical killer applications.

Despite the move toward a 32-bit environment, the 16-bit Windows 3.1 API will likely be secure for some time, making code portability increasingly important.

The whole scenario is about to be played out again for 32-bit apps, only the playing field is somewhat more level. The stable retail version of OS/2 2.0, along with a complete set of development tools, is already in the hands of tens of thousands of developers, some of whom have been subsidized by IBM to the tune of millions of dollars to generate shrink-wrapped OS/2 applications. OS/2 2.0 is being pitted against Microsoft's NT (New Technology) operating system and its WIN32 API, which first reached selected developers in fall 1991 and became broadly available to programmers in July 1992. The results of the test will be easy to evaluate. If, in spite of OS/2 2.0's head start, IBM feels obligated to add WIN32 support to a subsequent version of OS/2 in order to maintain the system's viability, we will know that the API issue was in fact a crucial one.

WIN16, WIN32, and WIN32S The whole concept of the WIN32 API has caused tremendous confusion among industry columnists and observers. Traditionally, the gold standard for a personal computer operating system API has been a particular *delivered version* of that operating system, that is, *what it does,* rather than the documentation (which was often incomplete, severely delayed, or downright misleading). When the behavior of the operating system and its documentation disagreed, the system itself always prevailed.

By contrast, the WIN32 API first arrived during early 1991 as a specification document—unaccompanied (and untainted) by any attempt at a real-world implementation. The specification drove the construction of the system, rather than being written as an afterthought. It's worth taking a few minutes now to clarify exactly how WIN32 and its cousin, WIN32S, are related to the Windows 3.*x* API.

For convenience, let's refer to the Windows 3.1 16-bit API—which includes the new functions for DDE (Dynamic Data Exchange), drag-and-drop, multimedia, common dialogs, and OLE (Object Linking and Embedding)—as the WIN16 API. This API reached its present configuration through a process of accretion; it is almost laughably asymmetric and contains a great many redundant or overlapping functions. It seems likely that the rich, complex, and wacky WIN16 API will be secure for several years—if for no other reason than that its object-oriented successor (called Windows Cairo by Microsoft) will be even more

hideously complex and will be [illegible] subject to many unforseen development delays and a lengthy beta test cycle.

WIN32 is an almost pure 32-bit superset of WIN16. In other words, for essentially every hardware-independent function in WIN16, there exists a function in WIN32 that has the same calling syntax and the same semantics. Because WIN32 is a superset of WIN16, though, WIN32 also includes many functions that don't exist in WIN16. These functions are designed to position WIN32 as a credible competitor to UNIX and the 32-bit OS/2 Presentation Manager API; they include support for security, networking, memory-mapped files, preemptive multitasking, threads, symmetric multiprocessing, Named Pipes, mailslots, semaphores, and advanced graphics engine capabilities such as paths, transforms, and Bezier curves.

The WIN32 API has been implemented initially on top of Microsoft's NT team led by ex-Digital Equipment Corp. operating system guru David Cutler, and its microkernel architecture is similar to Mach (a variant of Unix used on the NeXT computer). As I write this, NT/WIN32 is running on Intel 80386/486 and MIPS processors, and will soon be available for Digital's Alpha processor as well. Microsoft has also promised that WIN32 will be implemented on top of a "future version of DOS," though the company has been rather vague about the time frame.

Where does WIN32S fit into this picture? Apparently, WIN32S is Microsoft's attempt to counter widespread fears that NT will be too expensive, too powerful, and too resource-hungry for the average desktop machine. The *S* in WIN32S stands for subset: the WIN32S API consists of all the WIN32 functions—and only the WIN32 functions—that have direct counterparts in WIN16. WIN32S ap[illegible] mode applications, but [illegible] vantage over full-fledged WIN32 applications: The same binaries will run on either Windows 3.*x* or NT/WIN32 systems.

On an NT/WIN32 platform, a WIN32S application will not take advantage of all the NT capabilities, but it will benefit from NT's flat memory model, 32-bit graphics engine, and improved file system. On a Windows 3.*x* platform, each function call by the WIN32S application will be trapped by a translation layer in a DLL or Virtual Device Driver (VxD) and remapped into an ordinary WIN16 function call. The performance of WIN32S applications running under Windows 3.*x* will be throttled by the WIN16 graphics engine, but the ability to run on Windows 3.*x* will increase the short-term market for WIN32S applications dramatically, and should provide powerful motivation for developers to migrate their Windows programs from 16

WINAPP.C

1 of 2

```
// WINAPP - WIN16 and WIN32 Portable Application Skeleton
// Copyright (C) 1992 Ray Duncan
// PC Magazine * Ziff Davis Publishing

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements

#include "windows.h"
#include "winapp.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window

char szShortAppName[] = "WinApp";           // short application name
char szAppName[] = "Windows Demo App";      // long application name
char szMenuName[] = "WinAppMenu";           // name of menu resource
char szIconName[] = "WinAppIcon";           // name of icon resource

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeUINT messages[] = {
    WM_PAINT, DoPaint,
    WM_COMMAND, DoCommand,
    WM_DESTROY, DoDestroy, } ;

//
// Table of menubar item IDs and their corresponding functions.
//
struct decodeUINT menuitems[] = {
    IDM_EXIT, DoMenuExit,
    IDM_ABOUT, DoMenuAbout, } ;

//
// WinMain -- entry point for this application from Windows.
//
INT APIENTRY WinMain(HANDLE hInstance,
    HANDLE hPrevInstance, LPSTR lpCmdLine, INT nCmdShow)
{
    MSG msg;

    hInst = hInstance;                      // save this instance handle

    if(!hPrevInstance)                      // if first instance,
        if(!InitApp(hInstance))             // register window class
            return(FALSE);                  // exit if couldn't register

    if(!InitInstance(hInstance, nCmdShow))  // create this instance's window
        return(FALSE);                      // exit if create failed

    while(GetMessage(&msg, NULL, 0, 0))     // while message != WM_QUIT
    {
        TranslateMessage(&msg);             // translate virtual key codes
        DispatchMessage(&msg);              // dispatch message to window
    }
    TermInstance(hInstance);                // clean up for this instance
    return(msg.wParam);                     // return code = WM_QUIT value
}

//
// InitApp --- global initialization code for this application.
//
BOOL InitApp(HANDLE hInstance)
{
    WNDCLASS  wc;

    // set parameters for frame window class
    wc.style = CS_HREDRAW|CS_VREDRAW;       // class style
    wc.lpfnWndProc = FrameWndProc;          // class callback function
    wc.cbClsExtra = 0;                      // extra per-class data
    wc.cbWndExtra = 0;                      // extra per-window data
    wc.hInstance = hInstance;               // handle of class owner
    wc.hIcon = LoadIcon(hInst, szIconName); // application icon
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);       // default cursor
    wc.hbrBackground = GetStockObject(WHITE_BRUSH); // background color
    wc.lpszMenuName =  szMenuName;          // name of menu resource
    wc.lpszClassName = szShortAppName;      // name of window class

    return(RegisterClass(&wc));             //register class, return status
}

//
// InitInstance --- instance initialization code for this application.
//
BOOL InitInstance(HANDLE hInstance, INT nCmdShow)
{
    hFrame = CreateWindow(                  // create frame window
        szShortAppName,                     // window class name
        szAppName,                          // text for title bar
        WS_OVERLAPPEDWINDOW,                // window style
        CW_USEDEFAULT, CW_USEDEFAULT,       // default position
        CW_USEDEFAULT, CW_USEDEFAULT,       // default size
        NULL,                               // no parent window
        NULL,                               // use class default menu
        hInstance,                          // window owner
        NULL);                              // unused pointer

    if(!hFrame) return(FALSE);              // error, can't create window

    ShowWindow(hFrame, nCmdShow);           // make frame window visible
    UpdateWindow(hFrame);                   // force WM_PAINT message
    return(TRUE);                           // return success flag
}

//
// TermInstance -- instance termination code for this application.
//
BOOL TermInstance(HANDLE hinstance)
{
```

Figure 1: **This is the source code for the skeleton program that can be compiled for either WIN16 or WIN32.**

WINAPP.C
2 of 2

```
    return(TRUE);                           // return success flag
}

//
// FrameWndProc --- callback function for application frame window.
//
LONG FAR APIENTRY FrameWndProc(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    INT i;                                  // scratch variable

    for(i = 0; i < dim(messages); i++)      // decode window message and
    {                                       // run corresponding function
        if(wMsg == messages[i].Code)
            return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }
                                            // or hand off to Windows
    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoCommand -- process WM_COMMAND message for frame window by
// decoding the menubar item with the menuitems[] array, then
// running the corresponding function to process the command.
//
LONG DoCommand(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    INT i;                                  // scratch variable

    for(i = 0; i < dim(menuitems); i++)     // decode menu command and
    {                                       // run corresponding function
        if(wParam == menuitems[i].Code)
            return((*menuitems[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoDestroy -- process WM_DESTROY message for frame window.
//
LONG DoDestroy(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    PostQuitMessage(0);                     // force WM_QUIT message to
    return(0);                              // terminate the event loop
}

//
// DoPaint -- process WM_PAINT message for frame window.
//
LONG DoPaint(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;
    HFONT hfont;

    hdc = BeginPaint(hWnd, &ps);            // get device context
    GetClientRect(hWnd, &rect);             // get client area dimensions
    hfont = CreateFont(-36, 0, 0, 0, 700,   // get handle for pretty font
        TRUE, 0, 0, ANSI_CHARSET,
        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
        DEFAULT_QUALITY, (FF_MODERN << 4) + DEFAULT_PITCH,
        "Ariel");
    SelectObject(hdc, hfont);               // realize font in DC
    DrawText(hdc, "Hey, Dude!", -1,         // paint text in window
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    EndPaint(hWnd, &ps);                    // release device context
    return(0);                              // terminate the event loop
}

//
// DoMenuExit -- process File-Exit command from menu bar.
//
LONG DoMenuExit(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    SendMessage (hWnd, WM_CLOSE, 0, 0L);    // send window close message
    return(0);                              // to shut down the app
}

//
// DoMenuAbout -- process File-About command from menu bar.
//
LONG DoMenuAbout(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    WNDPROC lpProcAbout;                    // scratch far pointer

    lpProcAbout = MakeProcInstance((WNDPROC)AboutDlgProc, hInst);
    DialogBox(hInst, "AboutBox", hWnd, lpProcAbout);
    FreeProcInstance(lpProcAbout);
    return(0);
}

//
// AboutDlgProc -- callback routine for About... dialog
//
BOOL FAR APIENTRY AboutDlgProc (HWND hwnd, UINT msg, UINT wParam, LONG lParam)
{
    if((msg == WM_COMMAND) && (wParam == IDOK))
        EndDialog(hwnd, 0);
    else
        return(FALSE);
}
```

bits to 32 bits.

HOW BIG THE GAP? In the introduction to the *WIN32 Programmer's Reference Manual,* Volume I (Microsoft Press, 1991), Microsoft lists six general goals for the WIN32 API. The first two of these goals are to provide a 32-bit migration path for existing Windows applications and to make porting a Windows application to WIN32 as easy as possible. How completely has Microsoft achieved these goals? The same manual recounts "[The Windows 3.1] File Manager contains 20,000 lines of code; yet within one day, it was compiling as a native WIN32 application. Within a week, File Manager could execute and display directory listings." I don't find this to be an incredible feat at all; in fact, the admission that it took a whole week to get File Manager to execute as a WIN32 application makes reasonable the assumption that the original File Manager was very sloppily coded. In my own experience, clean Windows 3.*x* application code can be mechanically edited for WIN32 and will usually compile and run on WIN32 the very first time. The application code may not work entirely *correctly* the first time, but it will do something, and the trouble spots are normally quite obvious.

More importantly, if you study the WIN16 and WIN32 APIs carefully before writing your application, it's quite feasible to compile both WIN16 and WIN32 binaries from the exact same source code base. In fact, if you're willing to sacrifice the function calls that are unique to NT/WIN32—in other words, to settle for a WIN32S application instead of a full-fledged WIN32 application—you can readily hide all the differences between the two APIs with conditional compilation and #defines in the application's header files. All this certainly bodes well for the early availability of mainstream applications on NT/WIN32, and in fact Microsoft demonstrated Excel 3.0 running as a native WIN32 app before most Windows developers had even heard of WIN32. Let's take a minute to look at some of the differences between WIN16 and WIN32 that have to be provided for in the header file of such a "dual-targeted" application.

The most obvious discrepancy between the two APIs is that WIN16 was built for the Intel 80*x*86 segment architecture, while WIN32 was designed for the 32-bit flat memory model typical of the Intel 80386/486 running in paged virtual memory mode, the Motorola 680*x*0 family, and the RISC processors. Thus, the segment:offset *far* pointers of the WIN16 API become 32-bit *near* offsets in WIN32. Microsoft's 32-bit C compiler treats the FAR keyword essentially as a NOOP, and large-model WIN16 code (which doesn't attempt to take advantage of "magical" knowledge about the contents of a far pointer) will usually compile without changes for WIN32—though it will execute much more efficiently. Similarly, application callback functions are entered via near calls instead of far pascal calls in WIN32. The two calling conventions can easily be hidden within conditional #defines and made transparent to

WINAPP.H
Complete Listing

```
//
// WINAPP.H -- Header File for Demo Program WINAPP.C
//

#if !defined(WIN32)

  #define WIN16     TRUE
  #define WIN31

  #define INT       int
  #define UINT      WORD
  #define APIENTRY  PASCAL
  #define WNDPROC   FARPROC

#else

  #define WIN16     FALSE

#endif

struct decodeUINT {                          // structure associates
    UINT Code;                               // messages or menu IDs
    LONG (*Fxn)(HWND, UINT, UINT, LONG); };  // with a function

#define     IDM_EXIT     100
#define     IDM_ABOUT    101

// Function prototypes
INT APIENTRY WinMain(HANDLE, HANDLE, LPSTR, INT);
BOOL InitApp(HANDLE);
BOOL InitInstance(HANDLE, INT);
BOOL TermInstance(HANDLE);
LONG FAR APIENTRY FrameWndProc(HWND, UINT, UINT, LONG);
BOOL FAR APIENTRY AboutDlgProc (HWND, UINT, UINT, LONG);
LONG DoDestroy(HWND, UINT, UINT, LONG);
LONG DoPaint(HWND, UINT, UINT, LONG);
LONG DoCommand(HWND, UINT, UINT, LONG);
LONG DoMenuExit(HWND, UINT, UINT, LONG);
LONG DoMenuAbout(HWND, UINT, UINT, LONG);
```

Figure 2: **The #define statements in this header file supplement the WINDOWS.H header file to mask the differences between WIN16 and WIN32.**

a well-behaved application at both compile time and runtime.

Another point of divergence between WIN16 and WIN32 is that an integer in WIN16 code is, logically enough, 16 bits wide, while an integer in WIN32 is 32 bits wide. A "short" value, on the other hand, remains 16 bits in both systems and a "long" is likewise 32 bits in both systems. This means you must be very careful about using the proper data types for handles, logical and device coordinates, and other values that are defined as signed or unsigned integers in the WIN16 API, since these are all widened to 32-bits in the WIN32 API. For example, I often see Windows source code that declares parameters as WORD when they should be unsigned integers (UINT), or vice versa. When compiling 16-bit code, these data types are equivalent, but improperly specifying a WORD (16 bits) instead of UINT (32 bits) in WIN32 code will yield a compilation error if you're lucky, or a crash at execution time if you're not.

The third potential trouble spot—no, make that a *definite* trouble spot—is Windows messages. Windows applications are event-driven, and the events are represented as message packets that are posted to the application's message queue. The application responds to an event by reading a message from its queue and processing it; if the queue is empty, the application yields control of the system until a message arrives. A WIN16 message packet has four components: a 16-bit handle for the window that is to receive the message, a 16-bit message number (the possible values are defined in the Windows SDK header file as names of the form WM_xxxx), a 16-bit parameter called wParam, and a 32-bit parameter called lParam. In a WIN32 message packet, the four components have the same general usage, but all four are the same length: 32-bits.

Unfortunately, in the WIN16 API, certain messages used half of lParam for a handle of some type, and the remaining half of lParam for other data. Since WIN32 handles are 32 bits, such message packets obviously must be redefined for WIN32, usually by moving the "other" data from lParam to the upper or lower half of the widened wParam. Once you know about them, differences in message formats can be hidden by trivial conditional #defines in the header file, but the mutation of many important message packets is the main reason why newly ported WIN32 apps often only "sort of work" on first try after a clean compile.

This problem with message packets was really completely unnecessary. If Microsoft had elected to double the size of lParam along with everything else in

WINAPP.DEF
Complete Listing

```
NAME        WinApp

DESCRIPTION  'WinApp - Demonstration Windows Application'

EXETYPE     WINDOWS

STUB        'WINSTUB.EXE'

CODE        PRELOAD MOVEABLE
DATA        PRELOAD MOVEABLE MULTIPLE

HEAPSIZE    32768
STACKSIZE   8192

EXPORTS
  FrameWndProc
  AboutDlgProc
```

Figure 3: **This is the module definition file for the WINAPP program.**

the message packet, no redefinition of message formats would have been needed, and one of the few significant aggravations involved with porting applications to WIN32 would have been prevented. The decision to leave the size of lParam at 32-bits in WIN32 was predestined back when CL386 was being thrown together for the first OS/2 2.0 SDK. Ignoring history, Microsoft's languages division decreed that there was no need for their compiler to support true "longs" that were twice the size of a 32-bit integer because a 32-bit address or value was big enough for any app they could imagine. Less than four years later, with the very first prerelease version of NT/WIN32 running on a true 64-bit microprocessor (the MIPS R4000), I can't help but wonder whether those worthy language developers are still employed.

You've probably been wondering when I'd get around to talking about inconsistencies between the WIN16 and WIN32 APIs proper. The truth is, the glitches you'll encounter in this area will be extremely rare. In both WIN16 and WIN32, the same functions do the same hings with the same side effects and they use the same parameters. If structures are passed to or received from the functions, the elements of the structures have the same names and the same meanings. The one API-related circumstance that may cause some confusion occurs when the compiler generates error messages that refer to apparently nonexistent function parameters. What's almost always happening in such cases is that the function you think you're using is being remapped by a macro in the WIN32 header file to another (more general) function, and the parameters are being rearranged. Thus, for example, an invocation of CreateWindow() in your source code is silently remapped to a call to CreateWindowEx().

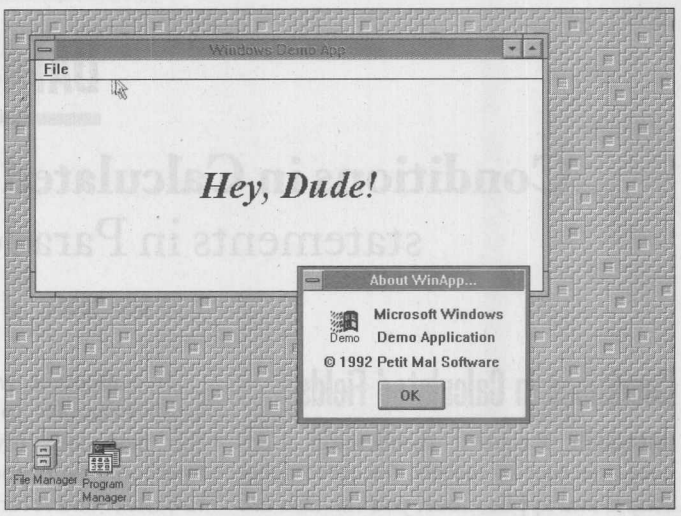

Figure 5: **This is a screen shot of the WINAPP program. Can you guess whether it's running under Windows 3.*x* or NT/WIN32?**

A PORTABLE APP SKELETON We will be discussing the WIN16, WIN32, and OS/2 PM APIs in more detail in upcoming columns, with particular attention to code portability. To give you some initial feeling for the similarities between the WIN16 and WIN32 APIs, however, I've written a skeleton application called WINAPP that can be compiled without changes for Windows 3.1 using Borland C++ 3.*x* or for NT/WIN32 with Microsoft's CL386. The source code for WINAPP can be found in Figure 1, the header file in Figure 2, the module definition file in Figure 3, the resource script in Figure 4, and a screenshot of WINAPP in action in Figure 5. All of these source files can be downloaded from PC MagNet. Or, if you don't have access to PC MagNet, you can get the files on-disk by sending a postcard with your name, address, and preferred floppy disk size to Katherine West, Power Programming, *PC Magazine,* One Park Ave., New York, NY 10016.

WINAPP is just a toy program of the "Hello World" genre, but it knows how to carry out almost all of the fundamental tasks of a Windows application. It processes messages, decodes menu commands, selects a font, paints in its window, and has a dialog box. The modularity and maintainability of the program is enhanced by use of the same table-driven message dispatcher that we previously saw in the DLGDEMO and SYSMON programs. You can use the WINAPP program as a starting point for much more elaborate applications that will run happily in either 16-bit or 32-bit mode; I've been doing it for quite a few months now, and you'll see some of the results in the next few issues.

THE IN-BOX Please send your questions, comments, and suggestions to me at any of these electronic mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan
INTERNET: duncan@csmcmvax.bitnet □

WINAPP.RC
Complete Listing

```
// Resource script for WINAPP.C

#include "windows.h"
#include "winapp.h"

WinAppIcon ICON winapp.ico

WinAppMenu MENU
BEGIN
    POPUP           "&File"
    BEGIN
        MENUITEM    "E&xit",    IDM_EXIT
        MENUITEM    SEPARATOR
        MENUITEM    "A&bout",   IDM_ABOUT
    END
END

AboutBox DIALOG 23, 19, 106, 61
CAPTION "About WinApp..."
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
BEGIN
    ICON "WinAppIcon", -1, 12, 9, 16, 16, WS_CHILD | WS_VISIBLE
    CTEXT "Microsoft Windows", -1, 33, 7, 64, 8
    CTEXT "Demo Application", -1, 30, 18, 68, 9
    CTEXT "\251 1992 Petit Mal Software", -1, 9, 30, 87, 9
    CONTROL "OK", IDOK, "BUTTON", WS_GROUP, 36, 43, 32, 14
END
```

Figure 4: **This is the resource script for the WINAPP program.**